Hydra: An Accelerator for Real-Time Edge-Aware Permeability Filtering in 65nm CMOS

Author(s):
Eggimann, Manuel; Gloor, Christian; Scheidegger, Florian; Cavigelli, Lukas Arno Jakob; Schaffner, Michael; Smolić, Aljoša; Benini, Luca

Publication date:
2018

Permanent link:
https://doi.org/10.3929/ethz-b-000310255

Originally published in:
https://doi.org/10.1109/ISCAS.2018.8351051

Hydra: An Accelerator for Real-Time Edge-Aware Permeability Filtering in 65nm CMOS

M. Eggimann†, C. Gloor†, F. Scheidegger†, L. Cavigelli†, M. Schaffner†, A. Smolic‡, L. Benini†
†ETH Zurich, Integrated Systems Lab IIS, Zurich, Switzerland ‡Trinity College, Dublin, Ireland

***Abstract*—Many modern video processing pipelines rely on edge-aware (EA) filtering methods. However, recent high-quality methods are challenging to run in real-time on embedded hardware due to their computational load. To this end, we propose an area-efficient and real-time capable hardware implementation of a high quality EA method. In particular, we focus on the recently proposed permeability filter (PF) that delivers promising quality and performance in the domains of high dynamic range (HDR) tone mapping, disparity and optical flow estimation. We present an efficient hardware accelerator that implements a tiled variant of the PF with low on-chip memory requirements and a significantly reduced external memory bandwidth (6.4× w.r.t. the non-tiled PF). The design has been taped out in $65\,\mathrm{nm}$ CMOS technology, is able to filter 720p grayscale video at $24.8\,\mathrm{Hz}$ and achieves a high compute density of $6.7\,\mathrm{GFLOPS/mm^2}$ (12× higher than embedded GPUs when scaled to the same technology node). The low area and bandwidth requirements make the accelerator highly suitable for integration into systems-on-chip (SoCs) where silicon area budget is constrained and external memory is typically a heavily contended resource.**

I. Introduction and Related Work

Edge-aware (EA) filters are important building blocks used in many image-based applications like stylization, high dynamic range (HDR) tone mapping, detail editing and noise reduction [1–12]. However, several high-quality methods such as the *weighted least squares* (WLS) filter [4] are computationally demanding and hence unsuitable for real-time applications on resource constrained devices. We focus on a recently proposed method termed *permeability filter* (PF) [11] that can be used to approximate image-based regularization problems such as HDR tone mapping [12], disparity [11], and optical flow estimation [13]. The PF has been designed to converge to results similar to the high-quality WLS filter, but with significantly lower computational effort [13] – which renders the PF an ideal candidate for high-quality filtering in real-time.

In this work, we present a hardware accelerator for the PF that can be used as an area-efficient co-processor in systems-on-chip (SoCs) tailored towards video processing. In particular, we contribute the following:

- We propose a tiled variant of the PF (TPF) with low on-chip memory requirements and a 6.4× lower off-chip memory bandwidth than the non-tiled PF.
- We devise an efficient hardware architecture for the TPF that employs loop pipelining and an optimized memory interleaving scheme. Our design maximizes floating-point (FP) unit utilization and eliminates memory contentions caused by frequent tile transpositions required by the TPF.
- We implement custom FP arithmetic on our accelerator to accurately filter feature maps involving HDR and coordinate data (e.g., sparse optical flow [13], [14]).
- Our design is the first custom hardware implementation taped-out in $65\,\mathrm{nm}$ CMOS technology, and provides a high compute density of $6.7\,\mathrm{GFLOPS/mm^2}$. When scaled to $16\,\mathrm{nm}$ technology, this is around 12× denser than in recent embedded GPUs. When applying 4 internal PF iterations the chip processes 720p monochromatic video at $24.8\,\mathrm{Hz}$ with a measured power of $445\,\mathrm{mW}$.

The PF approximates the high-quality WLS filter [4] with a low computational effort [11–13]. Furthermore, the PF features good halo reduction and information spreading capabilities that are important for HDR tone-mapping, regularization methods, sparse-to-dense conversions, disparity and optical flow estimation. Other EA filters such as variants of the *bilateral filter* (BF) [15] and the *guided filter* (GF) [6] are computationally less involved as the PF, but do not achieve the same level of quality and are hence used for different applications. We compare our chip with ASIC implementations of the BF [16] and GF [17] accelerators in Sec. IV.

II. Permeability Filter and Tiling

Similar to other EA filtering methods such as the GF and the *domain-transform*, the PF uses a *guiding image* $\mathbf{I}$ that controls the EA filtering behavior. The filtered data channels $\mathbf{A}$ may differ from the input image (e.g., in certain applications, $\mathbf{A}$ may hold other features like sparse optical flow vectors or disparity data). In a first step, the PF algorithm extracts *pairwise permeabilities* π^X_{pq} and π^Y_{pq} from the guiding image $\mathbf{I}$ [13]. Permeabilities measure the similarity between pixels at index p and q in the horizontal and vertical direction, and define the row-stochastic matrices H^X_{pq} and H^Y_{pq} holding the filtering coefficients for the horizontal and vertical filter passes. The PF is defined as a 1D operation over a *single row/column* in the image as follows:

$$J_p^{(k+1)} = \sum_{q=1}^{n} H_{pq} J_q^{(k)} + \lambda H_{pp}(A_p - J_p^{(k)}), \qquad (1)$$

where $\mathbf{A}$ holds the data channel to be filtered, $\mathbf{J}^{(k)}$ denotes the intermediate filtering result after k iterations ($\mathbf{J}^{(0)} = \mathbf{A}$), λ is a bias parameter towards the original data to reduce halo artifacts and n denotes the length of the current row/column to be filtered. To generalize the PF into two dimensions, we iteratively apply (1) on each row (called *X-Pass*) and column (called *Y-Pass*). Typical applications considered in this work (HDR tone-mapping [12], filtering of optical flow data [13]) apply $K = 4$ *XY-passes* to the entire frame.

Reformulating Equation (1) enables an efficient 1D scanline evaluation [11], [13]. Each *X-Pass* and *Y-Pass* is decomposed

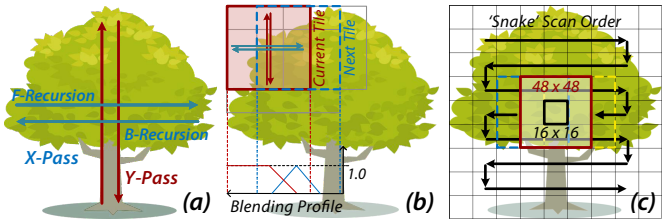

Fig. 1. (a) The PF filters the frame by alternating between horizontal and vertical 1D filtering passes over the whole frame. (b) The TPF splits the frame into overlapping tiles that are filtered individually. The individual results are merged together using a linear weighting profile. (c) Row-major 'snake' traversal scheme of the tiles.

into a forward and backward recursion. The forward recursion F_i with normalization weight $\hat{F}_i$ is given by:

$$F_p = \pi_{p-1}(F_{p-1} + J^{(k)}_{p-1}), \qquad \hat{F}_p = \pi_{p-1}(\hat{F}_{p-1} + 1.0), \quad (2)$$

and a backward recursion B_p with normalization weight $\hat{B}_p$:

$$B_{p-1} = \pi_{p-1}(B_p + J^{(k)}_p), \quad \hat{B}_{p-1} = \pi_{p-1}(\hat{B}_p + 1.0). \quad (3)$$

The $\boldsymbol{\pi}$ map in Equations (2) and (3) is either $\boldsymbol{\pi}^X$ or $\boldsymbol{\pi}^Y$ depending on the filtering direction. The initial values of the recursions are set to zero. Finally, the resulting filter output can be computed on-the-fly during the backward recursion by

$$J^{(k+1)}_p = \frac{F_p + J^{(k)}_p + B_p + \lambda(A_p - J^{(k)}_p)}{\hat{F}_p + 1.0 + \hat{B}_p}. \quad (4)$$

Hence, one PF iteration comprises a forward/backward recursion in x-direction, followed by a forward/backward recursion in y-direction, as illustrated in Fig. 1a.

A. Image Tiling

Since the PF alternates operating on rows and columns, the complete frame must be kept in the working memory. When applying the PF globally to one data channel $\mathbf{A}$ of a 720p frame ($h = 720$ and $w = 1280$) with a word width of $b = 24\,\text{bit}$ (see Sec. II-B for an evaluation) the required working memory amounts to $4 \cdot w \cdot h \cdot b \approx 88.5\,\text{Mbit}$ to hold the two permeability maps, $\mathbf{A}$ and $\mathbf{J}^{(k)}$ (without intermediate storage for $\mathbf{F}, \hat{\mathbf{F}}$). Since we are considering a co-processor scenario, such a large memory is unfeasible to be implemented on-chip, and hence requires off-chip memory. However, this results in a large off-chip memory bandwidth of $11 \cdot K \cdot 2 \cdot w \cdot h \cdot b \cdot \theta \approx 6.09\,\text{GB/s}$ for $K = 4$ filter iterations and a throughput of $\theta = 25\,\text{fps}$, which is not desirable. To this end, we propose a localized version of the PF, which operates on local square tiles as illustrated in Fig. 1b. To reduce tiling artifacts, we employ linear blending of neighboring tiles. By evaluating different overlap configurations, we found that overlapping tiles by 2/3 provides the best visual quality (see Fig. 2). With that configuration, nine different tiles contribute to result pixels. To minimize the memory requirements, we employ a 'snake' scan order (illustrated in Fig. 1c) to be able to reuse intermediate results for blending. Fig. 4 shows on-chip SRAM requirements caused by different tile sizes.

A larger tile size is desirable to better approximate the global filter behavior. The following considerations restrict the choice of the tile size: tiles should overlap by 2/3 edge lengths, the length must be divisible by three, and computing the linear weights for the final merging step is simplified when the length is divisible by a power of two. This results in a preferred tile size of $3 \cdot 2^l \times 3 \cdot 2^l$. We choose a tile size of 48×48 pixels, of which 32×48 pixels overlap the neighbouring tiles on each side. Using this tiling approach, the PF can be reformulated to Alg. 1, which can be implemented with only $4 \cdot 48^2 \cdot b \approx 27.6\,\text{kB}$ SRAM storage to hold one tile. Further, the external bandwidth, comprising the input data $\mathbf{A}$, $\boldsymbol{\pi}^X$, $\boldsymbol{\pi}^Y$, the filter output $\mathbf{J}^{(K)}$, and the partially blended tiles, reduces by $6.4\times$ to only $950\,\text{MB/s}$.

Fig. 2. (a) Filter result with an overlap of 1/2. (b) Filter result with an overlap of 3/5. (c) Filter result with an overlap of 2/3.

Algorithm 1 Scanline Permeability Filter

```
for all Tiles do
    for 1 : (2 × k) do
        for all Rows do
            Initialize recursions.
            Evaluate forward recursion.          ▷ Eq. (2)
            Evaluate backward recursion.         ▷ Eq. (3)
            Compute filter output.               ▷ Eq. (4)
        end for
        Transpose tile.
    end for
    Blend overlapping tiles (linear profile).    ▷ Fig. 1b.
end for
```

B. Numerical Precision

One use-case of the PF algorithm is to regularize sparse feature maps (e.g., optical flow vectors) and convert them to a dense representation. This operation requires high precision and dynamic that is difficult to handle with fixed-point arithmetic. On the other hand, single precision FP with full IEEE-754 support (denormals, NaN, etc) is not needed for this application. Fig. 5 shows an evaluation of different FP formats for dense image data, as well as for the optical-flow estimation procedure [13] that operates on sparse velocity data. Result quality is measured w.r.t. a double precision baseline with the peak signal to noise (PSNR) measure in the case of dense data, and with the average endpoint error (AEE) measure for sparse flow data. Exponents with 5 bit and below often lead to underflows for both data types and were hence not further considered. We chose to employ a 24 bit FP format (FP24) with 6 exponent and 17 mantissa bits in order to align the format to byte boundaries (a byte aligned 16 bit FP format would have led to unacceptable quality losses for both dense and sparse data). This leads to a negligible implementation loss below 2E-4 AEE for sparse flow data, and over 90 dB PSNR for dense image data.

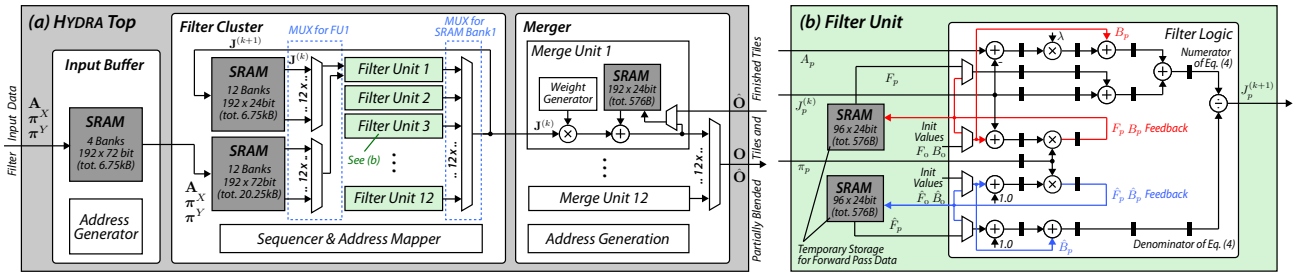

Fig. 3. (a) The HYDRA chip contains an input buffer, a filter cluster with 12 FUs, and a merger unit. (b) The systolic FU datapath implements the forward/backward recursions F_p, $\hat{F}_p$, B_p and $\hat{B}_p$, and employs pipeline interleaving to efficiently utilize the FP operators , as described in Sec. III-A.

III. ARCHITECTURE

Fig. 3a shows the proposed TPF architecture consisting of *input buffer*, *filter cluster* and *merger*. The *tiles* of the current frame are streamed through the *input buffer* into the *filter cluster* that operates on one 48×48 tile at a time. The *input buffer* aggregates the input such that it can be bursted into the *filter cluster* together with the last y-pass of the currently processed tile. I.e., the computation of the last y-pass is effectively overlapped with the data transfers that replace the now obsolete data in the two-port tile memories. During the last y-pass, the filter output is streamed into the *merger* unit that fuses the overlapping tile areas and finally outputs the results. The architecture implements the *tiled PF* with a fixed number of $K = 4$ iterations, and is parameterized to process monochromatic images with 720p resolution in real-time. However, the same design can be scaled to higher resolutions and multiple channels by deploying more parallel filter units (FUs). By minimizing the bandwidth to external memory, we maximize energy efficiency and facilitate integration into a larger system (e.g., as a filter accelerator in a mobile SoC).

A. Filter Cluster

Due to the filter feedback loop, FP24 units should operate at single cycle latency to achieve high utilization (Sec. III-B). Core frequencies up to 300 MHz achieve single cycle operation. Henceforth, we assume the limiting frequency of 300 MHz in the following throughput calculations. The 1D PF requires a single FP division per pixel. One 720p frame is split into 3354 tiles. Each pixel in a single tile is processed

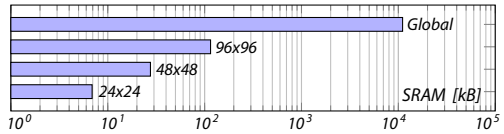

Fig. 4. Memory requirements for different tile sizes assuming 24 bit FP values (without temporary storage for intermediate results in the F, B recursions).

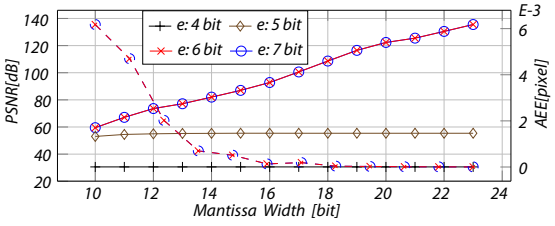

Fig. 5. PSNR (solid) and AEE (dashed) of the filter results for various floating point formats (exponent, mantissa width) compared to double precision.

8 times (4 iterations with 1 horizontal and 1 vertical pass each). In order to achieve a throughput of 25 fps, we need $25 \times 3354 \times 48^2 \times 8 \approx 1.55 \cdot 10^9$ FP divisions per seconds. Since the divisions are only performed during the backward recursion, we need at least 2×1.55 GFLOPS/300 MHz ≈ 10 FP dividers to run in parallel. The proposed architecture hence contains 12 parallel FUs. As described in the forthcoming sections, we employ pipeline interleaving and an optimized SRAM interleaving pattern to achieve a high utilization rate of 99.9% for FP multipliers and adders (49.9% utilization rate for the FP dividers), that is required to achieve the targeted throughput without further datapath replication.

B. Pipeline Interleaving

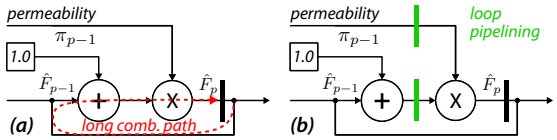

Fig. 6. Recursive part without (a) and with (b) pipeline register in the loop.

Fig. 6a shows part of the datapath inside the FUs. Due to the long combinational path through the FP adder and multiplier, timing closure can not be achieved at the target frequency of 300 MHz. The insertion of a pipeline register as in Fig. 6b improves the timing, but the feedback path from the multiplier back to the adder leads to a different functional behavior if the additional latency is not accounted for. To this end, a technique called *pipeline interleaving* is used [18]. Instead of processing a single row or column of the tile at a time, each FU simultaneously processes two lines in an interleaved manner. As can be seen in Fig. 7a, the next pixel from an even row enters the FU in even cycles, and the next pixel from the neighboring odd row enters the FU in odd cycles. With the additional pipeline stage the propagation delay of the critical path in the FUs can be reduced to 3ns.

C. Data Access Pattern

The proposed architecture provides simultaneous access to currently processed pixels in all operation modes and avoids filter pipeline stalls. To load twelve pixels in parallel, at least one memory block per filter block is required. Storing full rows of the tile in different memory blocks allows parallel access to the pixels during the horizontal pass but prevents simultaneous processing of the first pixels in the vertical pass since all the pixels of the first row reside in the same memory block. Instead, we employ an access pattern that subdivides

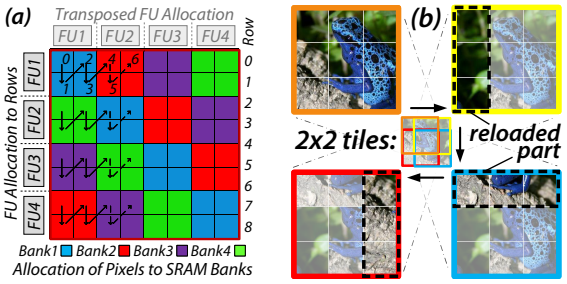

Fig. 7. (a) SRAM interleaving for stall-free tile transposition (shown for 4 FUs). (b) Tile fragmentation and cyclic replacement scheme. Note that stepping from one tile to the next only requires to reload 1/3 of the tile.

the tile into squares of 2×2 pixels denoted as $s_{i,j}$. The rule $m(s_{i,j}) = \mod (i+j, 120)$ assigns squares to memory blocks $\in [0, ..., 11]$. The square size of 2 pixels is motivated due to the *pipeline interleaving* and reduces the complexity of the address and memory index calculation units. The resulting checkerboard-like pattern is visualized in Fig. 7.

D. Cyclic Permutation of Pixel Locations

The proposed architecture tiles frames such that every two adjacent tiles overlap by 2/3, as explained in Sec. II-A. The tile size of 48×48 pixels implies – with the exception of the first tile of a frame – that 16×48 pixels (or 48×16 in the case of switching rows) need to be replaced. Reusing the remaining pixels reduces the input bandwidth. To maximize throughput, new pixels are stored where the now obsolete pixels were located in memory without reordering them. Since the individual tiles overlap by exactly $2/3$ the tile is subdivided into squares of 16×16 pixels (visualized in Fig. 7). The rows and columns of this 3×3 grid undergo a cyclic permutation that results in 9 different fragmentation states. The *filter cluster* and the *merger* keep track of the fragmentation state and transform the addresses accordingly. This approach increases the complexity of the address calculation but minimizes pipeline stalls in the *filter cluster* and allows 2/3 of the tiled pixels to remain in the SRAMs when stepping to the next tile.

IV. Implementation & Results

Our chip named HYDRA (depicted in Fig. 8) implements the proposed architecture and was fabricated in 65 nm CMOS technology. The design has been synthesised with Synopsys DC 2016.03 and P&R has been performed with Cadence Innovus 16.1. The total design complexity is 1.3 MGE of which 43% is occupied by the 52 SRAMs. HYDRA supports at-runtime configuration of the filter parameters and arbitrary video resolutions (with real-time performance up to 720p at 1.2 V). It also features a high FP24 compute density of $6.7\,\mathrm{GFLOPS/mm^2}$. When scaled to 16 nm, this would amount to $89\,\mathrm{GFLOPS/mm^2}$, which is around $12\times$ higher than in modern mobile GPUs manufactured in 16 nm. For instance, the NVidia Tegra X2 provides a theoretical peak throughput of 750 GFLOPS [19], and with an assumed silicon area of $100\,\mathrm{mm^2}$ for the GPU subsystem, this results in only $7.5\,\mathrm{GFLOPS/mm^2}$. In terms of external memory bandwidth, HYDRA requires 950 MB/s. This amounts to only 7.4 % of the total bandwidth provided by a LPDDR4-1600 64 bit

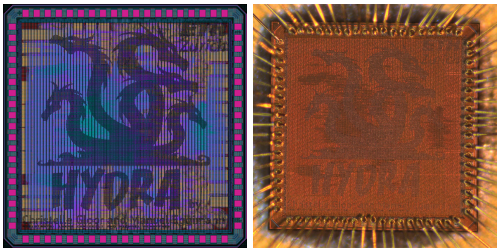

Fig. 8. CAD rendering and microphotograph of the HYDRA ASIC.

TABLE I
KEY FIGURES OF HYDRA AND COMPARISON WITH RELATED DESIGNS OPTIMIZED FOR APPLICATIONS THAT ONLY WORK ON 8 BIT SDR DATA.

	Properties \ Design	[16]	[17]	HYDRA
Algo.	Filter Type	Joint BF	GF	TPF
	Window Size [px]	31×31	31×31	48×48
	Arithmetic	FIXP	FIXP	FP24
Appl.	SDR Data	✓	✓	✓
	HDR Data	–	–	✓
	Sparse Coordinate Data	–	–	✓
Resources	Results from	Gate-Level	Post-Layout	Measured
	Technology [nm]	90	90	65
	Logic [kGE]	276	93	762
	SRAM [kB]	23	3.2	47.3
	Total Complexity [kGE]	-	-	1'328
Perform.	Resolution	1080p	1080p	720p
	Frequency [MHz]	100	100	259
	Throughput [fps]	30	30	24.8
	Core Power [mW/MHz]	-	0.23	1.72
	Bandwidth [MB/frame]	16.6	32.8	38

memory channel, which makes our design an ideal candidate for inclusion within a specialized domain accelerator in a SoC.

Tbl. I compares the key figures of HYDRA with two related accelerators [16], [17]. Note, that these designs implement simpler EA filters (BF/GF variants) with fixed point arithmetic since they have been developed to process dense, 8 bit standard dynamic range (SDR) data occurring in applications like flash denoising [17]. Our design is the only one that reports measured data, and it has been designed to support much more challenging HDR images and sparse optical flow data, which requires accurate arithmetic with high dynamic range (Sec. II-B). The PF provides better filtering quality than the GF and BF since it does not suffer from halo artifacts [12].

V. Conclusions

We present HYDRA, a compact and efficient accelerator for high-quality, permeability-based EA filtering in real-time. The accelerator employs a novel, tiled variant of the PF that significantly reduces the required on-chip memory and off-chip bandwidth compared to a non-tiled PF implementation. HYDRA is parameterized to deliver a throughput of 25 fps for monochromatic 720p video and provides significantly higher compute density than recent mobile GPUs. Our design is scalable to higher resolutions by increasing the amount of parallel FUs, and by employing higher-order pipeline interleaving to increase the operating frequency. Integrated into a SoC, the presented accelerator could act as a highly accurate and area-efficient co-processor for video processing applications.

REFERENCES

[1] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *ICCV*, Jan 1998, pp. 839–846.

[2] P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion," *IEEE TPAMI*, vol. 12, no. 7, Jul 1990.

[3] P. Milanfar, "A tour of modern image filtering: New insights and methods, both practical and theoretical," *IEEE SPM*, Jan 2013.

[4] Z. Farbman, R. Fattal, D. Lischinski *et al.*, "Edge-preserving decompositions for multi-scale tone and detail manipulation," *ACM TOG*, vol. 27, no. 3, pp. 67:1–67:10, Aug. 2008.

[5] R. Fattal, "Edge-avoiding wavelets and their applications," *ACM TOG*, vol. 28, no. 3, pp. 1–10, 2009.

[6] K. He, J. Sun, and X. Tang, "Guided image filtering," *IEEE TPAMI*, vol. 35, no. 6, pp. 1397–1409, June 2013.

[7] E. S. L. Gastal and M. M. Oliveira, "Domain transform for edge-aware image and video processing," *ACM TOG*, vol. 30, no. 4, 2011.

[8] E. S. Gastal and M. M. Oliveira, "High-Order Recursive Filtering of Non-Uniformly Sampled Signals for Image and Video Processing," in *Computer Graphics Forum*, vol. 34, no. 2, 2015, pp. 81–93.

[9] M. Aubry, S. Paris, S. W. Hasinoff *et al.*, "Fast Local Laplacian Filters: Theory and Applications," *ACM TOG*, vol. 33, no. 5, 2014.

[10] S. Paris, S. W. Hasinoff, and J. Kautz, "Local Laplacian Filters: Edge-Aware Image Processing With a Laplacian Pyramid," *ACM TOG*, vol. 30, no. 4, p. 68, 2011.

[11] C. Cigla and A. A. Alatan, "Information Permeability for Stereo Matching," *Signal Processing: Image Communication*, 2013.

[12] T. O. Aydin, N. Stefanoski, S. Croci *et al.*, "Temporally Coherent Local Tone Mapping of HDR Video," *ACM TOG*, 2014.

[13] M. Schaffner, F. Scheidegger, L. Cavigelli *et al.*, "Towards Edge-Aware Spatio-Temporal Filtering in Real-Time," *IEEE TIP*, vol. PP, 2017.

[14] M. Lang, O. Wang, T. Aydın *et al.*, "Practical Temporal Consistency for Image-Based Graphics Applications," *ACM TOG*, vol. 31, no. 4, 2012.

[15] F. Porikli, "Constant time o(1) bilateral filtering," in *IEEE CVPR 2008*, June 2008, pp. 1–8.

[16] Y. C. Tseng, P. H. Hsu, and T. S. Chang, "A 124 Mpixels/s VLSI Design for Histogram-Based Joint Bilateral Filtering," *IEEE TIP*, vol. 20, no. 11, pp. 3231–3241, Nov 2011.

[17] C. C. Kao, J. H. Lai, and S. Y. Chien, "VLSI Architecture Design of Guided Filter for 30 Frames/s Full-HD Video," *IEEE TCSVT*, vol. 24, no. 3, pp. 513–524, March 2014.

[18] H. Kaeslin, "Top-Down Digital VLSI Design, from VLSI Architectures to Gate-Level Circuits and FPGAs," *Morgan Kaufmann*, 2014.

[19] A. Skende, "Introducing parker next-generation tegra system-on-chip," in *2016 IEEE Hot Chips 28 Symposium (HCS)*, Aug 2016, pp. 1–17.